5th International Command and Control Research and Technology Symposium
Canberra Australia 24-26 October 2000

Interface Components for Coalition Interoperability.

Darryn J Reid
Land Operations Division
Defence Science and Technology Organisation
Department of Defence
The Commonwealth of Australia
PO Box 1500 Salisbury SA 5108
Australia.
Email: darryn.reid@dsto.defence.gov.au

Abstract

The problem of interfacing between a number of dissimilar coalition C4ISREW support systems is considered. In particular, it is proposed that the rapid development of reliable and extensible components to convey information between different systems can be achieved by employing a systematic and rigorous approach to defining the exact nature of the required interaction.

Varied formatted message languages, in particular, represent enormous barriers to interoperability, as they are typically difficult to assimilate by both man and machine. Distributed component technologies present a uniform connection model to the developer, yet they do not provide a complete solution in their own right to interoperability problems. The central premise of this paper is that dissimilarity in the semantic structure of the information represented by different systems is the fundamental issue that must be addressed in order for different systems to cooperate.

The behavior of the devices and software that provide interoperability should be justified in terms of the information structures that different participating systems represent. This may be achieved by separating the semantic structures from the details of how information is actually packaged and conveyed. The result of such an analysis is a specification of the behavior, in terms of these semantics, needed to convey information from one coalition system to another. A distributed component architecture then forms the framework within which syntactic descriptions, connection technologies, specifications of behavior, security constraints, and additional sources of supporting information can be brought together.

1. Introduction.

The requirement for interoperability between the diverse Command, Control, Communications, Computers, Intelligence, Surveillance, Reconnaissance and Electronic Warfare (C4ISREW) support systems used by a group of coalition partners necessitates the ability to bridge between a number of discordant system interfaces. These system interfaces might include formatted and unformatted message types, assorted database transactions, and object interfaces peculiar to various middleware platforms. All such interface types essentially represent languages, in the formal mathematical sense, meaning that they define a structure within which individual data values come together to form information.

Efforts aimed at providing system interoperability that approach each specific case as purely a software engineering problem have typically enjoyed only limited success, and at great cost. In contrast, this paper proposes a broadly applicable systematic approach for the resolution of arduous system interoperability problems, which emphasizes automated code generation, open architectures, and multiple levels of reuse.

The essential theme that underpins this paper is the that the heart of the problem of accomplishing interoperability between disparate systems lies not in different object standards, message formats, and connection mechanisms, but in fundamental disparities between the semantic structures that different systems encapsulate and present to one another. Furthermore, any claim of interoperability needs to be qualified in terms of the mappings between information structures embodied in the participating systems.

The overall approach advocates the separation of the information structures represented by a system interface from the details of how these structures are embodied in terms of language syntax and connection mechanisms. The many and varied functions necessary or helpful for interfacing between different systems in general are then categorized depending upon the range of their applicability. In particular, the information structures and the relationships between them directly define the most significant of these functions.

At an abstract level, the information structure of an interface of any coalition system can be described in terms of a set of relations between domains of data values, and the process of mapping between languages fundamentally involves interpreting the relationships between these structures. Conceptual modeling in database design is concerned with analyzing exactly these kinds of structures and the techniques suggested for utilization here are directly taken from this field.

When interpreted in the context of the distributed component methodology [Islam, 1996] [Pope, 1998] [Sigfried, 1993], the categorization of needed functionality reveals a core of flexible and reusable objects surrounded and supported by groups of other objects that are successively more specific in their applicability.

This paper focuses primarily on three types of components that are central to the proposed methodology. These are

- parsers, which reduce information received from coalition systems to a simple internal representation,
- translator engines, which process and manipulate information, and
- composers, which embody information in a form suitable for delivery to coalition systems.

The most significant of the core objects are the translator engines, which interpret behavioral specifications ultimately derived from the information structures represented within the system interfaces. Additional widely applicable services include naming services, security services, information repositories of various types, and factory objects that control and provide access to other objects of interest.

Various objects that are typically more specific to particular applications are also needed. These include parsers and composers for specific message languages, objects for bridging to other object domains, and objects capable of providing basic data communications with the coalition systems. It is particularly convenient to relate these objects with each other through inheritance of interface types, to form class hierarchies, and organized into libraries.

Together with their supporting middleware infrastructure these objects all form what will be termed the interface layer, to emphasize the fact that the mechanism through which coalition systems come together is internally composed of a host of interacting elements. The final members of the interface layer are the interface clients, which utilize the services provided by other objects in the layer to actually provide the interconnection between client systems. Interface clients are so named because they are clients of the services offered by other interface layer objects; in fact, as far as the coalition systems are concerned, the interface clients are actually service providers.

Trail implementations of the interface layer have been developed around various implementations of the Common Object Request Broker Architecture (CORBA) [OMG, 1999a] [OMG, 1999b] [Pope, 1998]. Other middleware backbones might also be used, including Sun Microsystems' Jini™, or the Distributed Computing Environment (DCE) [Brando, 1995] [Herman, 1992] [Vogel and Grey, 1995]. Applications of the methodology aimed at interfacing between a number of different simulations might be constructed instead using the US Department of Defence High Level Architecture [DMSO 1996a] [DMSO 1996b] [DMSO 1996c]. Other specific tools that have been used in experiments include the CLIPS expert system shell [Riley, 1990] and the Prolog logic programming language [Gal *et al.*, 1991] [Rowe, 1988].

2. Languages and information structures.

Different coalition systems to be brought together and made to cooperate typically present interfaces to the outside world that vary enormously in terms of syntax, connection technology and information structure. While all of these dissimilarities must be resolved to achieve interoperability, it is the information structure that is most the fundamental, and perhaps the most frequently overlooked. This section focuses on the development of models that describe such information structures independently of the details of how that information is packaged and communicated between systems.

The first step in this analysis is to construct conceptual models of information relating to individual coalition systems. The second step is to bring together models corresponding to coalition systems that are to directly communicate with one another. The final step in this process is to interpret the integrated schema or schemata to produce specifications of the behavior needed to produce information suitable for delivery to one system given information obtained from another.

It is often highly problematic to overcome the inconsistencies between the semantics of a large group of different systems to produce a single federated view to which all participants must comply. In contrast, the approach adopted here allows developers to choose anything from a federated approach, through integrating smaller groups of systems, to considering each individual pair of coalition systems separately.

The analysis of the interface provided by a coalition system segregates the coalition system interfaces into syntactic elements, semantic conceptual structures, and morphological configuration. Removed from the details of how it is actually implemented, the conceptual constitution of the information represented by a coalition system interface can be related on equal terms to that of a different system. The morphological structure is a simple index that directly relates syntactic elements to their conceptual symbolization.

2.1 Language Syntax.

The syntactic structure of many message languages may be conveniently expressed using Backus-Naur Form (BNF), Extended Backus-Naur Form, or some similar way of specifying a grammar [Barret *et al.*, 1986]. Many parser generator tools are available for interpreting or automatically producing code from such definitions. Figure 1 contains an EBNF syntax definition for a simple formatted message language, which is loosely based around some message specifications provided by the OTH-T Gold and ADFORMS formatted message standards [USN, 1996]. In the case of a coalition system that provides an object interface conforming to some distributed object standard, the syntax is simply definition of that interface.

```
Msg ::= MsgHeader {TargetSpecifier}+ EndOfMsg;
TargetSpecifier ::= AircraftGroup | VesselGroup;
AircraftGroup ::= AircraftSet {PositionSet}+;
VesselGroup ::= VesselSet {PositionSet}+;
MsgHeader ::= 'MSGID' '/' 'CONTACT' '/' ObserverId '//';
AircraftSet ::= 'AIRCRAFT' '/' TargetNumber '/' AircraftType '/' AircraftCategory '/' Country '//';
VesselSet ::= 'VESSEL' '/' TargetNumber '/' ShipName '/' ShipClass '/' ShipType '/' Country '//';
PositionSet ::= 'POSITION' '/' GeographicalPosition '/' DateTime '//';
EndOfMsg ::= 'ENDAT' '//';
```

Figure 1: Syntax definition for a simple formatted message language.

Enclosed in quotes are the literal terminal symbols of the grammar. Alternate possibilities are separated by vertical bars, and constructs enclosed in braces and appended with a plus sign may be repeated one or more times.

```
MSGID/CONTACT/yp783//
    VESSEL/vn509/arunta/anzac/ff/australia//
        POSITION/18.31s119.24e/210600091534//
        POSITION/18.17s120.53e/220600135216//
ENDAT//
```

Figure 2: An example of a simple formatted message.

The example message language shown here allows units to report the positions and identity of aircraft and ships they might observe. An example of a message conforming to this language, indicating the presence of the ANZAC class frigate HMAS Arunta at two locations west of Broome, Western Australia, is provided in Figure 2.

2.2 Semantic structures.

Specifying the behavior needed to interface between different coalition systems demands a proper analysis of the languages presented by those systems, and the relationships that bring them together. That is, the meaning of the information represented in both system interfaces must be expressed in some appropriate form. Database conceptual modeling languages might be used for this function; for example, Object-Role Modeling (ORM) [Halpin, 1994] and Entity-Relationship (E-R) diagrams [Elmasri and Navathe, 1989] are intuitively appealing graphical methods for expressing the relationships between various domains of data values. Other interesting possibilities that have been investigated include ontological approaches to conceptual modeling. In principle, any of these modeling paradigms may be used to establish conceptual schemata describing the information structures of the system interface languages [Reid and Davies, 1998].

An ORM conceptual schema describing this formatted message language is shown in Figure 3. The schema captures knowledge about the various kinds of data values that are represented in the language, and indicates how these values relate to each other. Also note that this schema might equally describe the information represented in an object interface definition, or in a database system, so that the approach is in no way specific to those systems that expect to communicate through formatted messages.

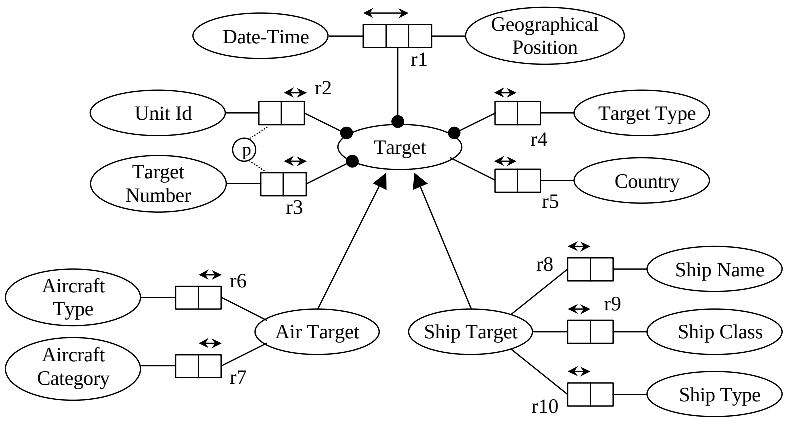

Figure 3: A simple conceptual schema.

Each ellipse (an ‘object type’) represents a set of data values which, in this case, may appear in a field of the example message type. The boxes (called ‘role boxes’) express the domains of mathematical relations between the elements of different sets, and the double arrows (‘uniqueness constraints’) above the role boxes indicate that the combination of values in the spanned roles must be unique. Relations, consisting of a number of role boxes together, are numbered for later reference. Heavy dots (‘mandatory role constraints’) at the base of the lines connecting an entity type with a role box indicate that every data value in the entity type must participate in the role.

For example, every target can have at most one related country of origin (r5), and every target must have at least one associated pair of position and date-time values (r1). The circled ‘p’ denotes the fact that each target is uniquely identified by the combination of the reporting unit and an assigned target number.

2.3 Schema Integration.

Once the schemata for the different system interfaces have been defined, they must then be integrated to produce a new conceptual schema that also embodies the relationships between structures from different schemata. Note that the subject of schema integration is the subject of much research [Batini *et al.*, 1986] [Colomb and Orlowska, 1994]

[Rusinkiewicz and Sheth, 1992] [Takizawa *et al.*, 1991], and cannot be properly addressed here. The precise schema integration techniques that will be used will depend upon individual preferences, the level of sophistication required, and the availability of suitable software support.

Broadly speaking, the process of schema integration involves a process of identifying common data sets (entity types in the ORM parlance), resolving conflicts caused by incompatible relationships, and making decisions about how other structures should be connected. As a simple example, consider another coalition system that uses unique identifiers for designating target contacts, rather than using target numbers that are only unique for a given observer. Integrating the conceptual schemata for these two interfaces would involve defining new relationships to relate the combination of observer and target number with unique target identifiers. One way of representing this is shown in Figure 4.

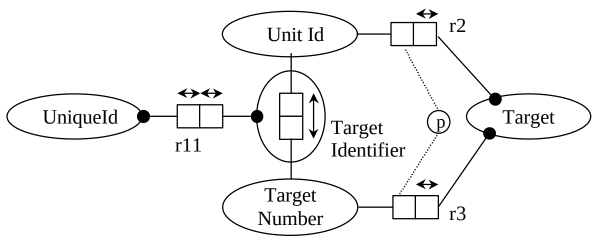

Figure 4: A fragment of an integrated schema.

The new ‘Target Identifier’ entity type is an objectified relation; each value in this set is a combination of unit identifier and target number values. The association of such combinations with unique identifiers is a simple one-one correlation, represented by the binary relation having uniqueness constraints on both roles. This mapping represents information that is not a part of either interface language, so that an implementation must include either data or computational procedures to, in effect, populate this relation with meaningful values.

To further highlight some of the issues that the schema integration process must address, suppose for the purpose of the argument that the interface presented by the second coalition system represents only ships, and does not know anything about aircraft. Converting information from this interface to conform to the requirements of the original interface presents no problem, but translating information in the opposite direction raises the possibility of receiving information that cannot be meaningfully conveyed. This highlights the possibility that only partial interoperability may be possible between many systems. One advantage of the overall approach is that any constraints and limitations on interoperability are unequivocally documented in terms of the information structures of the systems involved and the decisions made in integrating them.

Once the integrated schema has been developed, and a set of patterns defining the parts of the integrated schema that are populated by incoming information, the behavior required to convert from one language to another can be derived automatically. Such patterns, expressing which relations are populated when particular message types are received, or equivalently when specific object methods are remotely invoked, together form the morphology relating to the interface.

The reception of a new message of the kind described in Figure 1 implies that new values may populate the relations labeled r2 and r3 in the integrated schema of Figure 4. From this schema, it is clear that relation r11 is a one-to-one mapping between unit identifiers and target number pairs and the unique identifiers required in producing an output. Thus a production rule might be defined along the lines of the logic predicate example below.

Target(?unitId, ?tarNum, ?tarType) → ?uId=r11(?unitId, ?tarNum), Target(?uId, ?tarType).

The process of producing a behavioral specification suitable for use by a translator engine amounts to examining the possible sequences of operations that may be performed given each pattern of data values that might be received. Of these, sequences that produce the patterns of data values indicated by the morphology of another coalition system interface are of interest. In experiments on the automated interpretation of integrated conceptual schemata, Prolog [Gal *et al.*, 1991] [Rowe, 1988] has been used as a search engine to successfully find sequences of operations that transform input information into output. Further details of this work will appear in future publications.

3. The interface layer.

Interface clients provide the ability to inject the information contained in messages into database systems, compose messages from stored information, and translate information between different message languages or object structures. To maximize the opportunity for reuse of code and data, and to minimize the size and complexity of the entire system, an interface client is made up of individual message parsers, message composers, and translator engines chosen from libraries of such components. For example, a translator to map from one given language to another will utilize a single parser for decomposing the source messages, an engine for then manipulating the resultant information, and a message composer for formulating the appropriate final product. Experimental implementations of interface layer components have been built using CORBA [OMG, 1999a] [OMG, 1999b] [Pope, 1998]. Figure 5 illustrates the overall architecture of the interface layer, including parsers, composers, translator engines, a number of possible information sources, and the interface clients that actually convey information between coalition systems.

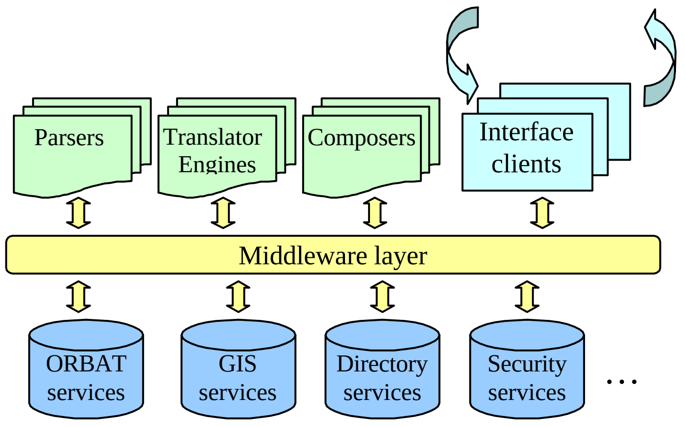

Figure 5: Schematic representation of the interface layer.

Thus the interface client itself is a very thin software module, merely linking together the parser, translator engine, and composer by performing a sequence of fairly simple remote method invocations. Its only other duties are to conform to the particular connectivity standards needed to receive data from the source systems and to deliver the results to the appropriate recipients. This itself might be supported by yet other components of the interface layer, such as directory services, mailboxes, and connection services.

Furthermore, translator engines are themselves broadly applicable and reusable components, being not specific to any particular interfacing application. Particular requirements for specific applications are embodied as a behavioral specification, which is submitted to and interpreted by the engine to achieve the necessary effect.

Parsers and composers might also interpret scripted definitions of required behavior, or could be generated using suitable compiler generation tools. Some parsers and composers might be hand-coded when other tools are inadequate for performance reasons or in the face of a particularly convoluted syntax. A combination of these approaches might also be considered.

3.1 Interactions between components.

Interface clients gain access to translator engines through one or more engine factory objects. In response to an appropriate request, the engine factory prepares an engine on behalf of the issuing interface client and in accordance with any provisions made as part of the request. Such provisions might include security and access constraints, details of the particular capabilities expected, and an indication of the behavioral specification to be used. Normally, and unless otherwise indicated, the translator engine will be bound exclusively to the particular interface client that issued the request, because the internal state of a translator engine at any moment potentially reflects the individual sequence of data submissions that have been received since the engine was created.

The engine factory interprets security and access constraints so that this restriction can be relaxed to permit certain cases in which multiple interface clients should be allowed to interact with the same engine, though this should only be done with care. An ability to define a translator engine as a resource shared by multiple interface clients could be useful when different clients represent non-overlapping information structures, or for providing a common repository for accumulating and fusing information from a number of sources. Even then, it is likely that the outcomes of an engine accepting information from multiple sources will still be directed to a single interface client. In this way, the translator engine essentially constitutes an information fusion node operating between several coalition systems.

The interplay between the various components within the interface layer is best achieved using a simple but sufficiently powerful internal format capable of supporting all possible information structures. That is, the result of parsing a message, the input to a composer, and both the input and output of a translator engine assume a common form, which can express instances of any possible conceptual relationship.

Logically, the information conveyed from one interface layer object to another is a set of elementary statements of relations between data values. These statements correspond with predicates in a logic programming language, facts to an expert system shell, or tuples according to the relational model of information, upon which relational database systems are based. Note this internal format is not a disguised assumption of a global schema, since the format does not place these simple statements against any particular structural constraints.

The scheme that has been used so far is particularly uncomplicated; the fact type precedes a simple list of assignments of particular values to attributes. The first line of Figure ?? illustrates how a single fact statement might be conveyed between interface layer objects.

The need to explicitly assign values to each attribute can be relaxed with the adoption of a convention for the order in which attributes are represented within the fact statement. This convention must be global, in the sense that all components that might receive a fact of a given type must recognize the same ordering of attributes. The bottom line of Figure 6 includes a second fact in the set in addition to an equivalent representation of the fact shown in the top line.

```
{ targetId(observer=02,tid=vn204,type=f18,category=f,country=aus) }

{ targetId(02,vn204,f18,f,aus), position(02,vn204,18.31s,119.24e,210600,091534) }
```

Figure 6: Some examples of sets of facts passed between objects.

This internal format for conveying information between interface components can be implemented in a host of different ways. For example, a flat string representation like that at the top of Figure 6 yields a particularly uncomplicated interface definition, but necessitates the use of small parsers and composers within each object to process such strings. A slightly more elaborate representation, such as that shown in the second part of Figure 7, eliminates the need to otherwise parse and compose character strings, but produces a somewhat more obscure object interface definition. This example code was constructed according to the OMG Interface Definition Language (IDL) [OMG, 1999a] [Pope, 1998] used by CORBA.

```
// A representation of sets of facts as a simple string.
typedef string FactList;

// An alternative structured data type for representing facts.
typedef sequence<string> ValueList;
struct Fact {
        string name;
        ValueList values;
        };
typedef sequence<Fact> FactList;
```

Figure 7: Two alternative representations of a set of facts.

Whichever kind of representation is chosen, it is important that object implementations do not attribute any particular significance to the order in which individual facts appear in the string or sequence.

3.2 Translator Engines.

A translator engine is an object which is capable of using knowledge of the relationships between different conceptual structures to derive new information conforming to one structure from given information that conforms to another. The knowledge needed to perform this function is supplied in the form of a program expressed in a high-level declarative language, produced by interpreting the integrated conceptual schema. Translator engines may also support other functions, such as the ability to communicate with other components, including other engines.

A simplified view of the internal structure of a translator engine is provided in Figure 8. Different kinds of inference control structures may be provided, including assertion-driven reasoning or forward chaining, goal-directed reasoning or backward chaining, and perhaps hybrid methods that combine the two [Hopgood, 1993] [Payne and McArthur, 1990] [Rowe, 1988]. The separation of the program interpreted by the engine into separate forward chaining production rules and backward chaining logic rules is merely notional, and this may or may not be reflected in any actual implementation.

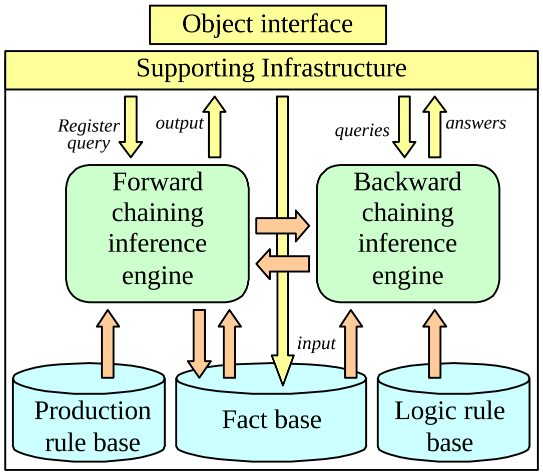

Figure 8: Simplified architecture of a translator engine.

Also illustrated are some of the interactions between different elements of the engine. Those that are named include information injected by an interface client or by another engine, the results that are sent back to the interface client, and requests to other components, usually for additional information that the engine itself requires but does not possess. All of this is hidden behind a simple object interface, through which all interactions with the outside world must occur.

```
interface Reciever {
        void recieveFacts(in FactList facts);
        void recieveError(in string message);
        };

interface Engine {
        void initialize(in string path, in Receiver interfaceClient);
        void submitFacts(in FactList facts);
        short registerQuery(in Query question, in Engine source);
        void withdrawQuery(in short handle);
        FactList executeQuery(in Query question);
        void release();
        };
```

Figure 9: A possible definition of an engine object interface.

Figure 9 illustrates how a translator engine interface might be defined in OMG IDL. Both exceptions and the parameters of methods that relate to access control have been omitted for clarity. The 'Receiver' interface is a callback mechanism to separate the delivery of information to the engine from the reception of its results; that is, the interface client that is to take delivery of the translated information must implement a 'Receiver' interface.

In its simplest form, a translator engine consists of a rule base for holding production rules and a fact base for storing fact statements, together with a forward chaining inference engine. Statements held in the fact base characterize actual instances of the relations between domains as expressed by the conceptual schema. The inference rules define actions to be taken when given conditions about the statements present in the fact base is true, and the forward chaining inference engine acts upon these rules to progressively modify the database and to deliver results back to the interface client. Traditional expert system shells support this kind of capability [Hopgood, 1993] [Payne, 1990]; in particular, CLIPS [Riley, 1990] has been used to construct some translator engines used in experiments.

Expert system shells do not provide the security, transaction management, or recoverability that may be required in a realistic application; on the other hand, commercial database products do offer these features. Most conceptual models can be mapped very easily onto a relational database schema, further suggesting that a relational or object-relational database might be an appropriate foundation for implementing a translator engine.

The preconditions and actions forming production rules can be expediently expressed using Structured Query Language (SQL) [Elmasri and Navathe, 1989]. Because the expert system shells are oriented towards manipulating individual facts, rather than complete tables, a production language built around SQL is significantly more expressive. Figure 10 illustrates how a rule written in a production language built around SQL might appear [Reid, 2000].

```
define formResult as
        select targetId, targetType, country from Targets, Ships
                where Targets.type = Ships.class
                group by targetId, country having count(*) = 1;
        insert into TargetResult values targetId, type, Country;
        delete from Targets;
end
```

Figure 10: An example of a production rule using SQL.

The precondition to the rule is expressed by a single SQL query that sets the context for the following sequence of data modification statements. The single rule shown here is an example that would require at least three rules to express in a normal expert system shell.

Alternatively, if an object-oriented conceptual schema were constructed, then an object-oriented database system might be a natural choice as the basis for building a translator engine. Preconditions and actions might then be expressed as a set of triggers and operations on database objects.

The statements held in the fact base fall loosely into two categories, depending upon how it is to be used. The first category of evanescent fact statements are those that exist only until they can be transformed into new statements; such facts form part of the normal flow of information through the engine. The second category consists of information that is not normally consumed in the translation process. Most of these facts in effect populate the parts of the integrated conceptual schema that connect structures corresponding to different language interfaces.

3.3 Information push and information pull.

The dynamic nature of the environment in which the interface client operates implies that this information held by an engine cannot remain constant. Some of this information may eventually become invalid, and new information may become available over time. Note that this base of stored information is distinguished from the normal flow of information between coalition systems only by the fact that the production rules that make use of it do not specify its removal. That is, new relatively persistent information is obtained through exactly the same mechanism used to receive transitory facts.

Through this mechanism, one translator engine can push useful information to other engines, provided that it possesses appropriate rules defining exactly what kind of information the others require. By adding methods to the translator engine interface through which other objects having the appropriate access permission can specify some kind of query and a return address, one engine may register with another its interest in particular kinds of information.

To elaborate further, the data passed to a translator engine requesting it to push particular kinds of facts to another engine is really a query, and implies the injection of a new production rule into the rule base. The query comprises the condition, or left-hand side, of this new rule, while the supporting infrastructure of the translator engine provides the action, or right-hand side. This action is actually very simple, only serving to pass the result of the query in an appropriate form back to the infrastructure, for delivery to the foreign translator engine.

A potential complication arises if the mappings from the structures represented on the conceptual schemata are not uniform for all translator engines. The normal rules presented to the forward chaining inference engine are constructed with intimate knowledge of the exact form of the kinds of facts to be found the in fact base. Another engine may not be in possession of such knowledge, if a different mapping has been used in its implementation.

As an example of this, consider again the conceptual schema of Figure 1. A 5th normal form relational database schema implementing this conceptual schema is illustrated in Figure 11. Each large box encloses the definition for a relational table, listing the attributes (or column names), with the key of the table being underlined. Optional attributes, which are those that are allowed to contain null values, are marked with an asterisk.

Targets			
UnitId	TargetNumber	TargetType	Country*

TargetPositions			
UnitId	TargetNumber	DateTime	GeoPosition

AirTargets			
UnitId	TargetNumber	AirType*	AirCategory*

ShipTargets				
UnitId	TargetNumber	ShipName*	ShipClass*	ShipType*

Figure 11: A 5th normal form relational schema.

This differs markedly from the simplest interpretation that might be envisaged, which would define separate tables for each relation of the schema (in this case, there would be nine binary tables and one ternary table). The reason for grouping certain relations into the same table relates to the possibility of anomalies that may otherwise be permitted, in violation particularly of mandatory role constraints on the schema.

Furthermore, not all components may even utilize a relational interpretation of the conceptual schema; other ways of implementation are also possible. Even the 5th normal form relational schema for a given conceptual schema is not, in general, unique. In fact, other 5th normal form relational schemata can be obtained for the example above, by dealing with the target subtype hierarchy differently. Figure 12 shows one such alternative, which was produced by effectively collapsing the air target entity into its parent.

TargetPositions			
UnitId	TargetNumber	DateTime	GeoPosition

Targets					
UnitId	TargetNumber	TargetType	Country*	AirType*	AirCategory*

ShipTargets				
UnitId	TargetNumber	ShipName*	ShipClass*	ShipType*

Figure 12: A different 5th normal form relational schema.

To concede to translator engines and other components the right to choose different implementation methods, a single interpretation of the conceptual schema might be selected to standardize on the form of the queries passed between interface components. It is then left to each engine to provide a way to map between the standard view and its own internal data structures; a backward chaining inference engine provides the capability needed to fill this gap.

Experiments conducted to date have considered only very simple kinds of query expressions, basically obtained by permitting variables in place of the data values within the fact statements of the internal format. For the purposes of illustration, assume that the 5th normal form relational schema of Figure 11 is adopted for communication between components. The first line of Figure 13 shows a simple query that would return the observers, target numbers, names and classes of all frigates known to the interrogated translator engine.

Some simple predicate patterns have also been supported, and the second and last line of Figure 11 briefly illustrates some of these. This query requests the details and locations of all aircraft that are either fighters or bombers, excluding F-18 and F-111 aircraft, which were observed by the unit identified as 251. The fairly simple scheme described here and used in experiments could, of course, be further extended as needed to permit even more complex queries.

```
{ shipTargets(?unitId, ?tarNum, ?name, ?class, frigate) }
{ targetPositions(251, ?tarNum, ?datetime, ?position),
        airTargets(251, ?tarNum,?airType:~f18&~f111,?airCat:fighter|bomber) }
```

Figure 13: Two examples of simple queries.

Note that this example is a little contrived; separate engines would usually communicate about relatively persistent information, rather than about such ephemeral facts, which are normally consumed in the translation process.

Suppose now that the 5th normal form relational schema of Figure 12 is used in the implementation of the translator engine to which these queries are submitted. The first of the queries is straightforward to satisfy, because the internal form does not differ from the standard against which queries are constructed. On the other hand, the second query requires an ability to convert between the external and internal representations, and a simple Prolog predicate that might be used to do this is shown below.

```
airTargets(UnitId, TarNum, Type, Cat) :- targets(UnitId, TarNum, air, _, Type, Cat).
```

In Prolog, variables begin with a capital letter, and an underscore indicates a variable whose actual value is not of interest. Although in this case the conversion is still rather straightforward, more complicated cases can be easily envisaged.

As well as supporting the ability of a translator engine to push information to other engines, the functionality outlined so far is immediately capable of supporting information pull. With a backward chaining inference engine to handle queries, another component can issue a request to have the query evaluated and any results returned.

Prolog has been used as the basis of an implementation of the translator engine; backward chaining is inherently part of the language, while the forward chaining capability can be provided programmatically. Like traditional expert system shells, Prolog does not provide the security, transaction management, and recoverability that might be required for realistic applications. The relational or object-relational database, while also addressing these issues, inherently provides the ability to map between the universally accepted external mapping and the particular table structures used internally through their support for views. That is, the requirement suggesting a backward chaining inference capability is automatically satisfied by the relational database view construct.

3.4 Other interface layer services.

Unlike translator engines, parser and composer objects do not generally need to preserve internal state information from one request to the next; a number of interface clients can therefore freely share a single parser or composer object. In these cases, the only restrictions that might be placed on access to parsers and composers would arise from security constraints that might imply a complete separation of information of different classifications. However, provided that a parser or composer is guaranteed not to preserve any state information between successive requests, the object needs only to be isolated while it processes the request. Moreover, the semantics of remote method invocation usually ensures such isolation.

The problem of parsing is ubiquitous in computer science, and there is a large range of tools available to support parser generation. These tools may equally be applied to producing composers, since the problem of composing an output message is actually one of parsing the internal format used to convey facts between interface layer components. Furthermore, the parsers and composers used in experiments were often implemented using the same expert system shells and logic programming languages utilized in building translator engines. This highlights the possibility that parsers and composers could, like the translator engine, interpret scripted programs expressed in high-level declarative languages. This approach was found to be particularly useful for building parsers and composers for subsets of natural language, which often appears in the free text fields of formatted message languages such as ADFORMS and OTH-T Gold.

Information from many different kinds of sources, such as Geographical Information Systems (GIS) and repositories containing Order of Battle (ORBAT) descriptions, may be required, depending upon the particular coalition systems to be interfaced. Other services that may be useful include directory services when information must to be redirected, depending upon its meaning or when live C4ISREW support systems are to be replaced with simulations. Security and access control services are also likely to be required; this is an important subject that cannot be fully discussed here.

Additional information that is useful or necessary for the translation, but not actually represented in the translator engine, might be also obtained from the coalition systems themselves. This possibility can be realized when the translator engine, acting through the interface client, can issue queries or register its interests with the coalition system. The information push and information pull capabilities already described are sufficient to support this scheme, provided that the coalition system interface provides the needed functionality and the interface client makes this visible to its translator engine.

4. Conclusion.

It is proposed that the behavior of any scheme purporting to provide interoperability must be verified and justified to ensure that it properly recognizes the information structures assumed by each of a number of dissimilar coalition C4ISREW support systems. Rather than focusing on particular implementations developed as part of this work, the emphasis here has been on the overarching concepts behind them. Specific strategies used for experimentation are described only to exemplify and support these concepts.

The paper has sought to highlight a number of elements that are fundamental in realizing interoperability between dissimilar coalition systems. The first of these elements is the formal representation of knowledge about the meaning of information received from or delivered to any participating system. In the proposed methodology, this factor is addressed by the development of conceptual schemata. The second element is knowledge about how the meaning of information from different systems is interrelated, which is encapsulated here within the process of integrating conceptual schemata. The third element is the application of such knowledge to then effect the translation. To address this issue, the integrated conceptual schema is interpreted in light of the patterns of structures that are populated by the coalition system interfaces, to produce a program that can be used by automated reasoning software.

The distributed component methodology is suggested as a basis for building suites of useful tools, in part because it explicitly advocates the separation of object interface from implementation details, and partly because it provides an open architecture to which new components can be added as required. In this way, different capabilities, built in different ways and by different development teams, can be readily brought together.

With the inclusion of additional components for supporting conceptual modeling, schema integration, parser and composer development, and for generating behavior specifications, the interface layer can support the system engineer at a very high level of abstraction. Future work will focus on the further development of such tools, and on the continued application of the methodology to interoperability problems on an increasingly larger scale.

References.

[Barret *et al.*, 1986] Barrett, W.A., Bates R.M., Gustafson D.A., Couch J.D., “Compiler Construction”, 2nd Ed, Science Research Associates Inc, 1986.

[Batini *et al.*, 1986] Batini C., Lenzerini M., Navathe S.B., “A comparative Analysis of Methodologies for Database Schema Integration”. ACM Computing Surveys Vol 18 No 4, 1986.

[Brando, 1995] Brando T., “Comparing DCE and CORBA”. Mitre Corporation Document MP95B-93, 1995.

[Brookes *et al.*, 1995] Brookes, W., Indulska J., Bond, A., Yang Z., “Interoperability of Distributed Platforms: a Compatability Perspective”, Proc. 2nd International Conf. On Open Distributed Processing, pp67-78, 1995.

[Colomb and Orlowska, 1994] Colomb R.M., Orlowska M.E., “Interoperability in Information Systems”, Information Systems Journal Vol 5, pp37-50, 1994.

[DMSO, 1996a] DMSO, “HLA Interface Specification Version 1.0”, Available at http://www.dmso.mil/projects/hla, 1996.

[DMSO, 1996b] DMSO, "HLA Rules Version 1.0", Available at http://www.dmso.mil/projects/hla, 1996.

[DMSO, 1996c] DMSO, "HLA Object Model Template Version 1.0", Available at http://www.dmso.mil/projects/hla, 1996.

[Edwards, 1999] Edwards, W.,K., "Core JINI", Prentice Hall, 1999.

[Elmasri and Navathe, 1989] Elmasri R., Navathe S., "Fundamentals of Database Systems", Benjamin/Cummings, Redwood City, Cal, 1989.

[Gal *et al.*, 1991] Gal, A., Lapalme, G., Saint-Dizier, P., Somers, H., "Prolog for Natural Language Processing", John Wiley & Sons, 1991.

[Halpin, 1994] Halpin T., "Conceptual Schema and Relational Database Design", 2nd Ed., Prentice-Hall, 1994.

[Herman, 1992] Herman J., "OSF's Distributed Management Environment", Business Comm. Review, Vol 22, No 5, pp59-63, 1992.

[Hopgood, 1993] Hopgood, A.A., "Knowledge-based Systems for Engineers and Scientists", CRC Press, 1993.

[Islam, 1996] Islam, N., "Distributed Objects: Methodologies for Customizing Systems Software", IEEE Computer Society Press, Los Alamitos CA, 1996.

[OMG, 1999a] Object Management Group, "The Common Object Request Broker: Architecture and Specification", revision 2.3, Document 98-12-01, June 1999.

[OMG, 1999b] Object Management Group, "CORBAservices: Common Object Services Specification", revision 2.3, Document 98-12-09, June 1999.

[Payne and McArthur, 1990] Payne, E.C., McArthur, R.C., "Developing Expert Systems", John Wiley & Sons, 1990.

[Pope, 1998] Pope, A., "The CORBA Reference Guide – Understanding the Common Object Request Broker Architecture", Addison-Wesley, Reading MA, 1998.

[Reid, 2000] Reid D.J., "Translating Deeply Structured Information", DSTO Technical Report DSTO-TR-0936, 2000.

[Reid and Davies, 1998] Reid D.J., Davies, M., "Towards a Gateway to Interconnect Simulations and Operational C3I Systems", Proc 3rd SimTecT98, pp 27-32, 1998.

[Riley, 1990] Riley, G., "CLIPS: An Expert System Building Tool", Proc 2001 Technology Conf., 1990.

[Rowe, 1988] Rowe, N.,C., "Artificial Intelligence Through Prolog", Prentice Hall, 1988.

[Rusinkiewicz and Sheth, 1992] Rusinkiewicz M.E., Sheth A.P., "Multidatabase Applications: Semantic and System Issues", Proc 18th International Conf. On Very Large Databases, 1992.

[Sigfried, 1993] Sigfried, S., "Understanding Object-Oriented Software Engineering", IEEE Press, Piscataway NJ, 1993.

[Takizawa *et al.*, 1991] Takizawa M., Hasegawa M., Deen S., "Interoperability of Distributed Information Systems", Proc. 1st International Workshop on Interoperability in Multidatabase Systems, pp239-242, 1991.

[USN, 1996] US Navy Center for Tactical Systems Interoperability, "Operational Specification for Over-the-Horizon Targeting Gold", Rev. B Ch. 2, 1996.

[Vogel and Grey, 1995] Vogel A., Grey B., "Translating DCE IDL in OMG IDL and vice-versa", DSTC Technical Report 22, 1995.